Law Office of
HOLLAND & KNIGHT LLP

701 Brickell Avenue
Suite 3000
Miami, Florida 33131
Telephone (305) 789-7773

Application for United States Letters Patent

filed on behalf of

Applicant(s): Laxmi P. Parida
For: System and Method for Encoding and Detecting Extensible Patterns
Attorney Docket: YOR920030163US1

SYSTEM AND METHOD FOR ENCODING AND DETECTING EXTENSIBLE PATTERNS

CROSS-REFERENCE TO RELATED APPLICATIONS

Not Applicable.

STATEMENT REGARDING FEDERALLY SPONSORED-RESEARCH OR DEVELOPMENT

Not Applicable.

INCORPORATION BY REFERENCE OF MATERIAL SUBMITTED ON A COMPACT DISC

Not Applicable.

FIELD OF THE INVENTION

The invention disclosed broadly relates to the field of information processing systems, and more particularly relates to the field of systems for detecting patterns in information strings.

BACKGROUND OF THE INVENTION

A rigid motif is a repeating pattern in a string of data comprising a plurality of tokens such as alphabet characters, possibly interspersed with don't-care characters that has the same length in every occurrence in the input sequence. Pattern or motif

discovery in data is widely used for understanding large volumes of data such as DNA or protein sequences.

Allowing the motifs to have a variable number of gaps (or don't-care characters), called patterns with spacers or extensible motifs, further increases the expressibility of the motifs. For example, given a string s = abcdaXcdabbcd, m=a.cd is a rigid pattern that occurs twice in the data at positions 1 and 5 in *s*. In the above example, the extensible motif, where the number of don't-care characters between *a* and *c* of the pattern is one or two, would occur three times at positions 1, 5 and 9. At position 9 the dot character represents two gaps instead of one.

The task of discovering patterns must be clearly distinguished from that of matching a given pattern in a string of characters or database. In the latter situation, we know what we are looking for, while in the former we do not know what is being sought. Typically, the higher the self similarity (i.e., repeating patterns) in the sequence, the greater is the number of patterns or motifs in the data. Motif discovery on data such as repeating DNA or protein sequences is indeed a source of concern because these exhibit a very high degree of self similarity. The number of rigid maximal motifs could potentially be exponential in the size of the input sequence.

The problem of a large number of motifs is usually tackled by pre-processing the input, using heuristics, to remove the repeating or self similar portions of the input or using a statistical significance measure. However, due to the absence of a good understanding of the domain, there is no consensus over the right model to use. Thus there is a trend towards model-less motif discovery in different fields. There has been

empirical evidence showing that the run time is linear in the output size for the rigid motifs and experimental comparisons between available implementations.

Consider an example of a rigid pattern (albeit with don't-care characters) discovery tool that is often inadequate in detecting biologically significant motifs. Fibronectin is a plasma protein that binds cell surfaces and various compounds including collagen, fibrin, heparin, DNA, and actin. The major part of the sequence of fibronectin consists of the repetition of three types of domains, which are called type I, II, and III. The type II domain is approximately forty residues long, contains four conserved cysteines involved in disulfide bonds and is part of the collagen binding region of fibronectin. In fibronectin the type II domain is duplicated. Type II domains have also been found in various other proteins. The fibronectin type II domain pattern has the following form:

C...PF.[FYWI].....C-(8,10)WC....[DNSR][FYW]-(3,5)[FYW].[FYWI]C

The extensible part of the pattern is shown as integer intervals. It is clear that a rigid pattern discovery tool will never capture this as a single domain. Therefore, there is need for a system and method of pattern discovery that overcomes the drawbacks in the prior art.

SUMMARY OF THE INVENTION

Briefly, according to the invention an input string of tokens (e.g., characters) is analyzed for token patterns. A system embodying the invention is based on an

inexact suffix tree construction which provides a framework for an output sensitive detection algorithm.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 shows an inexact suffix tree for a token string.

FIGs. 2a-e are a flow chart illustrating a method according to the invention.

FIGs. 3-14 illustrate construction of an inexact suffix tree for a string comprising the character sequence axcdabydaxy.

FIGs. 15-16 are tables showing the results of using a method according to the invention on a collection of fibronectin sequences.

FIG. 17 is a small sample output after using the method according to the invention on a collection of fibronectin sequences.

FIG. 18 is a block diagram of an information processing system using the invention.

DETAILED DESCRIPTION

To facilitate a clear understanding of the present invention, definitions of terms employed herein will now be given.

Dot character: The '.' is called a "don't-care" or a dot character and any other element is called solid. Also, σ will refer to a singleton character or a set of characters from $\sum$. Let s be a sequence of sets of characters from an alphabet $\sum$, '.' $\notin$ $\sum$. For brevity of notation, a singleton set is not enclosed in curly braces. For example, let $\sum = \{A, C, G, T\}$, then $s_1 = ACTGAT$ and $s_2 = \{A,T\}CG\{T, G\}$ are two

possible sequences. The j^{th} $(1 \leq j \leq |s|)$ element of the sequence is given by s[j]. For instance in the above example $s_2[1] = \{A,T\}$, $s_2[2] = \{C\}$, $s_2[3] = \{G\}$ and $s_2(4) = \{T, G\}$. Also, if x is a sequence, then $|x|$ denotes the length of the sequence and if x is a set of elements then $|x|$ denotes the cardinality of the set. Hence $|s_1| = 6$, $|s_2| = 4$, $|s_1|[1] = 1$ and $|s_2|[4] = 2$.

$e_1 \underline{\prec} e_2$: We say that $e_1 \underline{\prec} e_2$ if and only if e_1 is a don't-care character or e_1 is a subset of e_2. For example, if $e_1 = \{A, C\}$, $e_2 = \{A,C,G\}$ and $e_3 = \{T\}$ are three elements of some sequence, then $e_1 \underline{\prec} e_2$ and $e_1 \not\underline{\prec} e_3$.

Annotated Dot Character, $.^{\alpha}$: An annotated "." character is written as $.^{\alpha}$ where α is a set of non-negative integers $\{\alpha_1, \alpha_2, ..., \alpha_3\}$ or an interval $\alpha = [\alpha_l, \alpha_u]$, representing all integers between α_l, and α_u including α_l, and α_u. To avoid clutter, the annotation superscript α will be an integer interval.

Rigid, extensible string: Given a string m, if at least one dot element, is annotated, m is called a extensible string, otherwise m is called rigid.

Realization: Let m be a extensible string. A rigid string m' is a realization of m if each annotated dot element , $.^{\alpha}$ is replaced by l dot elements where $l \in \alpha$. For example, if $m = a.[^{2,4}]b.[^{3,6}]cde$, then $m' = a...b...cde$ is a realization of m and so is $m'' = a...b....cde$.

m occurs at l: A rigid string m occurs at position l on s if $m[j] \underline{\prec} s[l + j)$ holds for $l \leq j \leq |m|$. A extensible string m occurs at position l in s if there exists a

realization m' of m that occurs at l. If m is extensible then m could possibly occur a multiple number of times at a location on a string s. For example, if $s = axbcbc$, then $m = a.[^{1,3}]b$ occurs twice at position 1 as ***axbcbc*** and *axbc**bc***.

Size of m, |m|: If m is rigid, size of m is the number of solid and dot characters in m and is denoted by $|m|$. If m is extensible and occurs at positions in L_m, then $|m| = \max_i |m'_i|$ where m'_i is a realization of m that occurs at $i \in L_m$. Consider $s = abcdabeed$. Let $m_1 = ab$, $m_2 = ab\text{-}d$. Then $|m_1| = 2$ and $|m_2| = \max\{|ab.d|, |ab..d|\} = 5$.

k-*motif m, location list* L_m: Given a string s on alphabet $\sum$ and a positive integer k, $k \leq |s|$, a string (extensible or rigid) m is a motif with location list $L_m = (l_1, l_2, ..., l_p)$, if $m[1] \neq$ '.', $m[|m|] \neq$ '.' and m occurs at each $l \in L_m$ with $p \geq k$. Also L_m is complete, i. e., if there exists j such that m occurs at j then $j \in L_m$. To avoid clutter, in the rest of the discussion a k-motif will be referred to simply as a motif. The associated k should be clear from the context.

Realization of a motif m of s: Given a motif m on an input string s with a location list L_m, and m' a realization of the string m, then m' is a realization of the motif m if and only if there exists some $i \in L_m$ such that m' occurs at i in s.

Notice that because of our notation of annotating a dot character with an integer interval (instead of a set of integers), not every realization of the extensible string occurs in the input string. For example for $s = axbcbc$, $p = a^{[1,3]}b$ is a extensible motif on s. $p' = a..b$ is a realization of the string p but not of the motif p since p' does not occur in s. In the remaining discussion we will use this stricter definition of motif realization unless otherwise specified.

$m_1 \preceq m_2$: Given two motifs m_1 and m_2 on s $m_1 \preceq m_2$ holds if at each occurrence i on s, the realization m'_1 of motif m_1 at i there exists a realization m'_2 of motif m_2 at i such that $m'_1[j] \preceq m_2[j]$, $1 \leq j \leq l$ where $l = \max |m'_1|, |m'_2|$. For example, let $m_1 = AB..E$ and $m_2 = ABC.E.G$ occurring at position 1 of string s = ABCXEYGXXABYYEABCYEYG. Then $m_1 \preceq m_2$ at position 1, and $m_2 \preceq m_1$ at position 1.

Sub-motifs of motif m: Given a motif m let $m[j_1], m[j_2], \ldots m[j_l]$ be the l solid elements in the motif m. Then the sub-motifs of m are given as follows: for every j_i, j_t, the sub-motif $m[j_i \ldots j_t]$ is obtained by dropping all the elements before (to the left of) j_i and all elements after (to the right of) j_t in m.

Maximal Motif: Let $m_1, m_2, \ldots, m_k$ be the motifs in a string s. A motif m_i is maximal in composition if and only if there exists no m_l, $l \neq i$ with $L_{mi} = L_{ml}$., and $m_i \preceq m_l$. A motif m_i, maximal in composition, is also maximal in length if and only if there exists no motif m_j, $j \neq i$, such that m_i is a sub-motif of m_j and $|L_{mi}| = |L_{mj}|$. A maximal motif is maximal both in composition and in length.

Cell: Given s, a cell is the smallest substring in any pattern on s, that has exactly two solid characters: one at the start and the other at the end position of this substring.

$m^s_1 \succ m^s_2$: Given two cell m^s_1 and m^s_2, $m^s_1 \succ$ if one of the following holds:

1. m^s_1 has only solid characters and m^s_2 has at least one non-solid character
2. m^s_2 has the "-" character and m^s_1 does not

3. m^s_1 and m^s_2 have d_1, $d_2 > 0$ dot characters respectively and $d_1 < d_2$

Clearly, the above defines a partial order on the cells.

⊙-compatible, $m_1 \odot m_2$: m_1 is ⊙-compatible with m_2 if the last solid character of m_1 is the same as the first solid character of m_2. Further if ml is ⊙-compatible with m_2, then $m = m_1 \odot m_2$ is the concatenation of m_1 and m_2 with an overlap at the common end and start character and

$$L'_m = \{((x,y),z) \mid ((x,l),z) \in L'_{m1}, ((l, y), z) \in L'_{m2}$$

For example if $m_1 = ab$ and m_2 = b.d then m_1 is ⊙-compatible with m_2 and $m_1 \odot m_2 = ab.d$. However, m_2 is not ⊙-compatible with m_1.

Fixed vs. Variable Spacers

We now discuss two different kinds of spacers, fixed and variable. Given a constant D, for rigid motifs it is to be interpreted that the motif can have fixed 1 or 2 or ... or D dots between successive solid characters and for extensible motifs can have between 1 to D dot characters between successive solid characters. Also, let R be the set of all rigid maximal motifs and let ε be the set of all extensible motifs. The following statement can be easily verified. Given a string s with parameters k and D, then if $m_r \in R$, then there must be $m_f \in \varepsilon$ such that either $m_r = m_f$ or m_r is a substring of m_r.

Consider the following two examples. Example 1: If $s = aycazxc$ with $k = 2$, $D = 2$, then $\varepsilon = \{a\text{-}c\}$ and $R = \{\}$ with $|\varepsilon| > |R|$. Example 2: Let $s =$

abycpqdefabzcxdef with $k = 2$, $D = 2$. Then $R = \{ab.c, de\,f\}$ and $\varepsilon = \{ab.c\text{-}def\}$ with $|R| > |\varepsilon|$. Thus, although in theory there is no relationship between $|R|$ and $> |\varepsilon|$, in practice $|R| < |\varepsilon|$.

Inexact Suffix Tree

We introduce a data structure representing a string, called an inexact suffix tree that efficiently stores all the suffixes of maximal patterns with wild cards (variable or don't-care tokens). The suffix is inexact in the sense that it contains wild cards.

Referring to FIG. 1, there is shown an inexact suffix tree for a string s = axcdabydaxy and with $D = 2$. A solid circle denotes a leaf node. The root node is labeled Z and the internal nodes are labeled A through I. The unique path from the root node (Z) to the leaf node labeled by integer i, represents a string $p \prec s[i \ldots n]$.

The inexact suffix tree is built as follows: Given a string s of size n, let $\$ \notin \Sigma$. We terminate s with \$ as s\$. Let D be the maximum number of don't care characters between any two consecutive solid characters and let k be the minimum number of times a extensible pattern must occur. Consider a rooted tree T with edges labeled by non empty strings with the following properties:

Each leaf node is labeled by an integer $1 \leq l \leq n$. The edge label is a sequence on $\Sigma + \{\text{'.'},\text{'-'}\}$. All the outgoing edges of the root node are labeled by strings that start with a solid character. No two edges out of a node can be labeled with strings that start at the same solid character. Each edge label can have at most D consecutive '.' character and at most one consecutive '.' character. Also, the last

character must be solid. For an internal node i, let $R(i)$ be the set of integer labels of the leaves reachable from i.

(a) $|R(i)| > 1$ for all i.

(b) For any two immediate successor internal nodes j_1 and j_2 of i, $R(j_1) \neq R(j_2)$.

(c) Consider an internal node i and its immediate successor j and let $p = x_1x_2 \ldots x_j$, where $x_i \notin \Sigma$ or x_i is the don't-care or '–' character, be the label on edge from i to j.

i. $R_j \subset R_i$.

ii. Consider all possible labels $p_1, p_2, \ldots p_l$ satisfying the constraint 2 above with each having J characters and the same R_j, then $p_1, p_2, \ldots p_l \preceq p$.

iii. There does not exist label p' satisfying all the constraints above with p' having less than J characters and a possible successor node j' of i such that $R_j \subset R_{j'}$, and $|R_j| > 1$.

It is easy to now see that given D, the inexact suffix tree is well defined and is unique. When D = 0, τ is also called a suffix tree of s. The tree described above is for $k = 2$ for clarity of exposition. It can be trivially generalized to $k > 2$.

The unique path from the root node to the leaf node labeled by integer i, represents $s[i \ldots n]$ that is obtained by traversing from the root node to the leaf node: concatenating the edge labels of this path gives p and $p \preceq s[i \ldots n]$.

The string associated with the internal node E is p = a..da.y obtained by concatenating the labels on the edges from the root node Z. Given strings p_i, $1 \leq i \leq l$, their meet is p if and only if $p \preceq p_i$, and there exists no p' such that $p \preceq p' \preceq p_i$, for all i. We make the following observations about the inexact suffix tree described above. The string obtained by concatenating the edge labels on the unique path from the root

to an internal node corresponds to a suffix of a maximal extensible pattern with parameter D and k=2.

Equivalently, consider the internal node j and let p be the string obtained by concatenating the labels on the edges in the path from the root node to the node *j*. Then p is the meet of the suffixes *s[i . . . n] where* $i \in R(j)$.

The inexact-suffix tree not only suggests a way of detecting all the extensible patterns efficiently but also gives a data structure for storing the extensible patterns for efficient retrieval or matching.

Implementation

An inexact suffix tree is constructed implicitly (in a different order) in an implementation that is discussed herein. Notice that the suffix tree, described above and illustrated in FIG. 1, produces all the suffixes of the maximal motifs. In the implementation, we detect the suffixes as early in the process as possible and discard them.

Referring to FIG. 2a, in step 201 we receive an input of a string *s* of size *n* and two positive integers, *k* and *D*. We begin with a few definitions that will be used in this step. Notice that a cell is the smallest extensible component of a maximal pattern and the string can be viewed as a sequence of overlapping cells. If no don't care characters are allowed in the motifs then the cells are non-overlapping.

The algorithm comprises the following steps:

In step 203 of FIG. 2a, we begin by constructing patterns that have exactly two solid characters in them and separated by no more than D spaces or "." characters. This can be done by scanning the string s from left to right. Further, for each location we store start and end positions of the pattern. In the following example a character could also be referred to as a cell, and a string of characters can also be referred to as cells.

For example, if s = abzdabyxd and $k = 2$, $D = 2$, then all the patterns generated at this step are: ab, a.z, a..d, bz, b.d, b..a, zd, z.a, z..b, da, d.b, d..y, a.y, a..x, by, b.x, b..d, yx, y.d, xd, each with a list of where they occur. Further $L_{ab} = \{(1, 2), (5, 6)\}$, $L_{a.z} = \{(1, 3)\}$ and so on.

In the next step 205, we construct the extensible cells by combining all the characters with at least one dot character and the same start and end solid characters. In step 207 we update the location list to reflect the start and end position of each occurrence. In the previous example, we generate *b-d* at this step with $L_{b\text{-}d} = \{(2, 4), (6, 9)\}$.

In decision 209, we determine whether the number of times a pattern repeats is less than k. Next in step 211, if $|L_m| < k$, then we discard all extensible strings m. Continuing the previous example, the only surviving cells are *ab, b-d* with the following equation:

$$L_{ab} = \{(1, 2), (5, 6)\} \text{ and } L_{b\text{-}d} = \{(2, 4), (6, 9)\}$$

This step is used only for quick comparison of location lists and is not vital for the working of the algorithm. Nevertheless, in practice it is a time saving step in detecting non maximal motifs (suffixes of maximal motifs).

In step 213 the input sequence s is broken up into $l \geq 1$ subsequences, called zones, $Z_1 = s[1, j_1]$, $Z_2 = s[j_1 + 1, j2]. . . , Z_l = s[j_{l-1} + 1, jl]$ $Z_{l+1} = s[j_l + 1, |s|]$ such that each occurrence of each cell is fully contained in a subsequence. This works best when 1 is much smaller than n.

We now continue the previous example, $l = 1$ with $z_1 = 4$. Thus Z_1 = abzd and Z_2 = abxyd. In step 215 we associate the zone number with every occurrence of the cell, and add each occurrence to a collection of cells B, this is continued as step 217 checks all the subsequences of the input sequence have been associated with zones. Thus the augmented location lists are

$$L'_{ab} = \{((1, 2), 1), ((5, 6), 2)\} \text{ and } L'_{b\text{-}d} = \{((2, 4),1), ((6, 9), 2)\}$$

We make the following statements about the zones that is straightforward to verify.

Given s, if an occurrence of a cell m^s_i contained in a zone Z_i then the occurrence of a corresponding maximal extensible pattern m_i that contains this occurrence of m^s_i is also contained in Z_i. Hence, the corresponding occurrence of every nonmaximal extensible pattern w.r.t m_i is also contained in Z_i.

Consider a maximal extensible motif m_i with $|L_{mi}| = K$. Let the K-tuple of m_i of only the zone numbers be defined as $Z_{mi} = (z_1, z_2, \ldots, z_k)$ Then for every nonmaximal motif m'_i w.r.t. m_i the following holds: $Z_{m'i} = Z_{mi}$

Iteration Phase.

We begin with a few definitions that will be used in these steps which are used in an iteration function described below. Let *B* be a collection of extensible strings. If m = Extract (B), then $m \in B$ and there does not exist $m' \in B$ such that $m' \succ m$ holds.

The iteration function will have an input of a collection of extensible strings *B* and an extensible string *m'* extracted from the collection of extensible strings *B* and an output of maximal extensible patterns. The output of maximal extensible patterns will be added to a collection of maximal extensible patterns called *Result*. The following definition is a reiteration of the order of the nodes described in constructing the inexact suffix tree, but stated in terms of cells for clarity of exposition.

The procedure is best described by the following pseudocode.

```
Result ←{};                          Iterate(m,B,Result)
B←{m^s_1 | m^s_1 is a cell};         {
                                     G:1   m'←m;
For each m = Extract(B)              G:2   For each b = Extract(B) with
 Iterate(m,B,Result);                G:3   ((b⊙-compatible m') OR ((b ⊙-compatible b))
 Result ← Result ∪ B                 G:4       If (m' ⊙-compatible b)
                                     G:5          mt ← m '⊙ b;
                                     G:6          If siblingInconsistent (m_i) exit;
```

G:7 If ($|L_{m'}| = |L_b|$) $B \leftarrow B - \{b\}$;

G:8 If ($|L_{m'}| \geq k$) $m' \leftarrow m_t$

G:9 If (b $\odot$ *compatible* m')

G:10 $m_t \leftarrow b \odot m'$

G:11 If SiblingInconsistent(m_i) exit;

G:12 If ($|L_{m'}| = |L_b|$) $B \leftarrow B - \{b\}$;

G:13 If ($|L_{m'}| \geq k$) $m' \leftarrow m_t$

G:14 **Iterate** ($m', B, Result$)

G:15 For each $r \in Result$ with $Z_r = Z_{m'}$

G:16 If (m' is not maximal w.r.t. r) return ;

G:17 Result $\leftarrow$ Result $\cup$ {m'};

}

Steps G:15-16 detect the suffix motifs of already detected maximal motifs. *Result* is the collection of all the maximal extensible patterns.

Referring to FIG 2b, in step 219 we extract an extensible string m from a collection of extensible strings B.

In the next step 221, we create a rigid string m' from the extracted extensible string m.

In the next step 223, we extract another extensible string b from the collection of extensible strings B.

Next in step 225, we determine whether the rigid string *m'* is compatible with the extensible string *b,* or if the extensible string *b* is compatible with the rigid string *m'*. If it is determined that they are not compatible with each other, then we replace the extensible string *b* with another extensible string extracted from the collection of extensible strings *B*.

In step 227, if both the rigid string *m'* and the extensible string *b* are compatible with each other, then they are concatenated to form a new extensible string m_t.

In step 229, we then check if the concatenated extensible string m_t is non-maximal with respect to its earlier siblings by checking the location lists. This routine corresponds to backtracking, which is always only when the sibling has don't care characters in it.

In step 231, if the concatenated string m_t is non maximal, then the method exits to the next iteration (exits the loop).

Referring to FIG 2c in step 233, if the concatenated string extensible string m_t is maximal with respect to its siblings, then we check if the number of times the rigid string *m'* repeats is equal to the number of times the extracted string *b* repeats.

In step 235 if the number of times the rigid string *m'* repeats is equal to the number of times the extracted string *b* repeats is true, then we remove the extracted string *b* from the collection of extensible strings *B*.

In step 237 if the number of times the rigid string *m'* repeats is equal to the number of times the extracted string *b* repeats is false, or after we remove the extracted string *b* from the collection of extensible strings, we determine if the number of times the rigid string *m'* repeats is greater than or equal to the parameter value *k*.

If the number of times the rigid string *m'* repeats is not greater than or equal to the parameter value *k*, the method returns to step 233 and repeats the iteration.

In step 239 if the number of times the rigid string *m'* repeats is greater than or equal to the parameter value *k*, then we convert the concatenated extensible string m_t to a rigid string and replace the rigid string *m'* with the converted concatenated string m_t. The iteration function in G:14 repeats recursively until collection of cells B is empty. Result is the collection of all the concatenated strings extracted. Result is updated whenever a non-maximal pattern is found.

In step 241 we continue to function G:15 when we cannot find any more strings b to concatenate with m.

Referring to FIG. 2d step 243, determine if the concatenated rigid string *m'* is not maximal with respect to the extracted pattern *r*.

If the concatenated rigid string *m'* is not maximal with respect to the extracted pattern *r*, then we return to G:15.

In step 244 we determine if there are any remaining patterns *r* in the collection of results *Results* to extract.

In step 245 if there are no remaining patterns *r* in the collection of results *Results*, then we add the concatenated rigid string *m'* to the collection of results *Results*.

If there are remaining patterns *r* in the collection of results *Results*, then continue to extract from the collection of results *Results* by returning to step 241. Once the iteration function has completed, we continue by constructing an inexact tree from the collection of results *Results*.

In step 247 we begin by extracting the first pattern from the collection of results.

In step 249 we create a root node from the first character in the extracted pattern.

Next, in step 251, we then continue by ordering lower level nodes from left to right of the root node starting with the patterns with no dot characters on the left, to the patterns with up to the parameter *D* number of dot characters. This step generates a tree with a single lower level, as shown in FIG. 3.

In step 253 of FIG. 2e, then perform a depth first traversal of each node starting with the left most node and continuing to the right. This step is illustrated in FIG. 4 for the first node on the left and continues as shown in FIG. 5 with the next node to the right.

The tree construction involves some limited backtracking. In step 255, the backtracking is always only one level deep and it occurs when the edge label has don't care characters in it. For example if the edge label is ".a", then the other siblings of this node are examined to see if the don't-care character is required. This step is illustrated in FIG. 6.

In step 257 we eliminate the identified node from the tree if the backtracking identifies an edge to the left which already contains the pattern. Otherwise we keep the node and perform a depth first traversal as in step 253 as shown in FIG. 7 and FIG. 8. FIG. 9 and FIG 10 also show a node that does not get eliminated because of sibling inconsistencies.

Step 259 determines if there are any remaining nodes to be checked for inconsistencies. If so, then we continue to step 261, which checks the next node to the right. If there are no further nodes that need to be checked we continue to step 263, which removes all edges that lead to leaf nodes, which is shown in FIG. 11.

Step 265 checks if there are nodes remaining that have more than one outgoing edge. FIG. 13 shows the rightmost node has an outgoing edge.

In step 267 if there are nodes with more than one outgoing edge then the outgoing edge is consolidated to a single outgoing edge, as shown in FIG. 14.

Step 269 completes the construction of the inexact suffix tree. FIG. 14 is a complete inexact suffix tree.

Allowing motifs to have a variable number of gaps (or don't-care characters), i.e., patterns with spacers or extensible motifs, considerably increases the expressibility of the motifs. It is likely that some information missed by rigid motifs is captured by the extensible motifs. The invention is an implementation of an extensible motif discovery algorithm that guarantees the detection of every extensible pattern. One of the directions being currently investigated is to use the method according to the invention to detect extensible patterns in an unsupervised manner on protein sequence databases, and then use suitable pruning techniques to compare the detected patterns with known motifs.

Referring to FIG. 15, there is shown a table showing the output of a small sample, wherein the input data is a collection of fibronectin sequences with $D = 7$ and $k = 2$. The first column gives the number of occurrences of the motif shown in the third column; the second column gives the number of distinct sequences in which the motif appears and the last column gives the occurrence in the format $(s : i_1 \quad i_2)$, where s is the sequence number and the motif starts at i_1 ending at i_2.

Referring to FIG. 16, there is shown a table wherein the input data is a collection of fibronectin sequences with $D = 7$ and $k = 2$. A small sample output is shown in the table. The first column gives the number of occurrences of the motif shown in the third column; the second column gives the number of distinct sequences in which the motif appears. This version uses homologous grouping of the amino acid bases shown in square brackets. The occurrence lists have been removed to avoid clutter.

Referring to FIG. 17, there is shown a table wherein for a small sample output, the input data is a collection of fibronectin sequences with $D = 7$ and $k = 2$. Here the gaps are annotated: -(i_1, i_2), which indicates that the number of gaps are between $i_1$and i_2 in the occurrences in the input.

The system of FIG. 18 is useful on primarily biological data such as DNA and protein sequences. However the generality of the system makes it equally applicable in other data mining, clustering, and knowledge extraction applications. The system comprises an input/output device 1806, a CD Drive 1808, a central processing unit 1802, and a memory unit 1804. The memory unit 1804 further comprises an operating system 1812, and an application 1814. The input/output device further comprises a network interface 1807.

Therefore, while there has been described what is presently considered to be the preferred embodiment, it will be understood by those skilled in the art that other modifications can be made within the spirit of the invention.

What is claimed is: